Microsoft Research Short Course

COM+ Runtime Technology (Part One)

Jim Miller
Program Manager,
COM+ Runtime

COM+ Short Courses

- Today: "VOS level"
  - Type System
  - Metadata
  - Intermediate Language
  - JIT Compilers
- Next week: "Enterprise"
  - Garbage Collection
  - Assemblies
  - Contexts
  - Remoting
- Proposed: Security
- Proposed: What's wrong with the COM+ Runtime

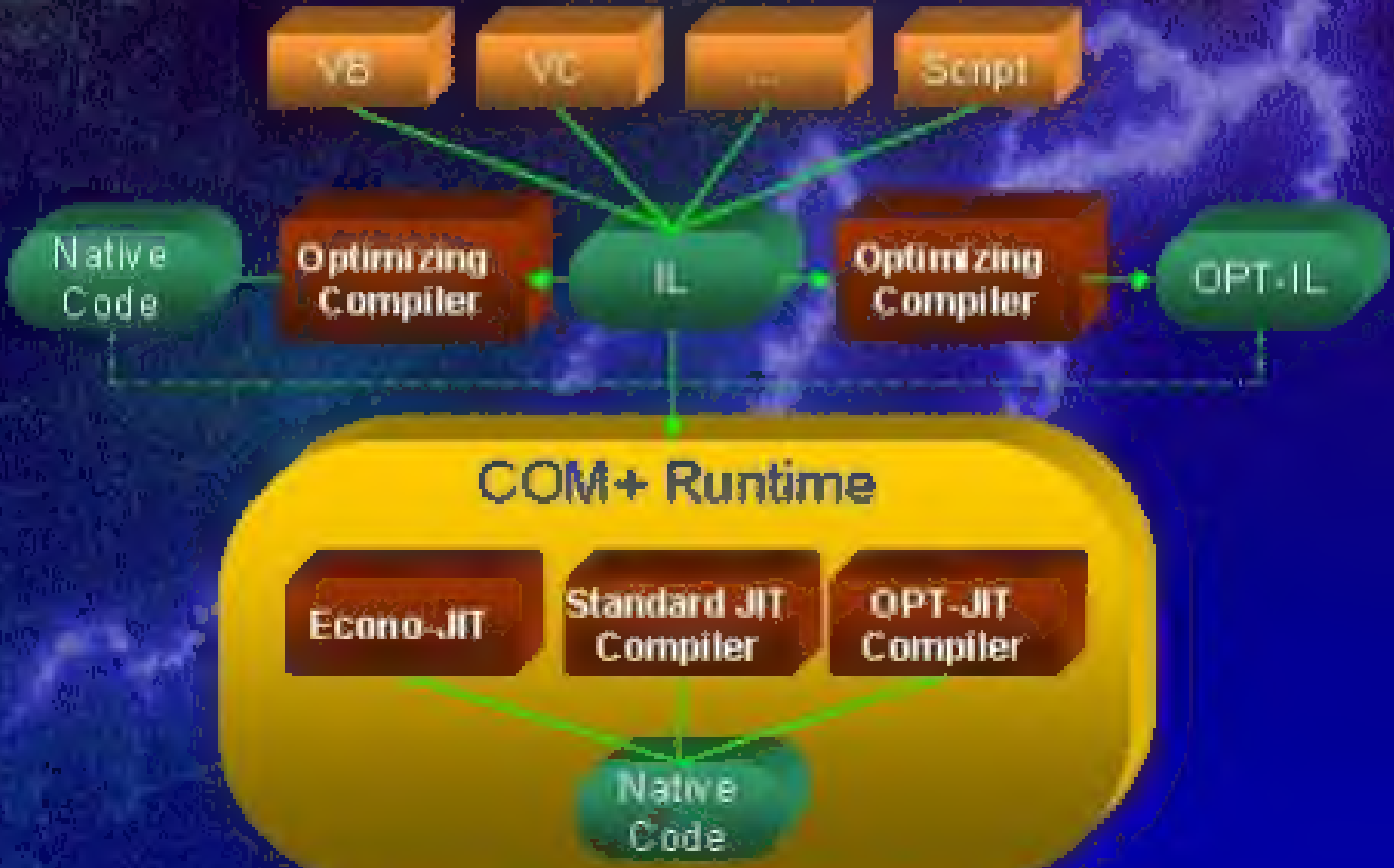
Agenda

- Architecture Overview
- Type System
- Metadata
- Intermediate Language (IL)
- Just-In-Time Compilers (JITs)

Architectural Overview

- Lowest level is Microsoft IL + objects
- Language support is built over execution and object model
- Some languages assume reflection, base classes
- COM classic interop assumes classes, reflection, deployment
- COM+ Services are combination of COM classic and runtime objects
- WFC and Win+ will leverage COM classic interop

IL Execution



Agenda

- Architecture Overview
- Type System
  - http://comrtime/specs/VOS/VOS Spec.doc
- Metadata
- Intermediate Language (IL)
- Just-In-Time Compilers (JITs)

Type System

- Types and Locations
- Type Members
- Naming
- Contracts and Signatures
- Type Definers
- Inheritance, Implementation, and Contractual Obligations

Types and Locations

- Value Types
  - User-defined types (like complex numbers, nullable integers)
  - "Sequences of bits," no inheritance hierarchy
  - Usually passed by value, equality but not identity
  - Can be "boxed" into Object Types
- Reference Types
  - Object types are self-describing at runtime, includes inheritance
  - Interface types are partial descriptions, usually implemented by many object types
  - Pointer types are machine addresses, possibly with static type, but without runtime description
- Locations
  - Hold values (i.e. instances of value types or reference types)
  - Have static signature (type, constant, read-only, volatile, etc.)

Type Members

- Fields and methods
- Mostly-orthogonal Attributes
  - Static + Instance + Virtual
  - Parent + Child
  - Private + Public + Restricted
  - Final + Not Final
  - Value Type + Interface + Class
  - Field + Method
  - Compilation + Module + Assembly + Global

Contracts and Signatures

- **Contracts**
  - Class contract: methods, fields, properties, events
  - Interface contract: methods, static fields, properties, events
  - Method contract
  - Property contract
  - Event contract
- **Signatures (enforceable part of contract)**
  - Values: constant
  - Locations: type + read-only, volatile
  - Parameters: location + by-ref, typed reference
  - Methods: parameters + varargs

Type Definers

- Classes created automatically by COM+ Runtime based on description provided by tool
- Arrays
  - Description includes element type, rank, lower bounds, and/or bounds
  - Methods include Length (total number of elements), GetLength, GetLowerBound
- Pointers
  - Description is data type of destination, including signatures for function pointers

Inheritance, Implementation, and Contractual Obligations

- Value types don't inherit (unless boxed)
- Inheriting means implementing the interface contracts
- Implementing an interface, property, or event implies fulfilling the corresponding contract
- Single inheritance only, but an object can implement multiple interfaces, properties, and events
- Thus, an object may implement many contracts (and we could say "an object is of many types")

Agenda

- Type System
- Metadata
  - http://comruntime/specs/Formats/signatures.doc
  - http://comruntime/specs/Formats/COR Metadata Interfaces.doc
  - http://comruntime/specs/ClassLibs/Runtime/Reflection/default.htm
- Intermediate Language (IL)
- Just-In-Time Compilers (JITs)

Metadata

- Overview
- Components and Assemblies
- What's in the Metadata
- Signatures and Binding
- Access to Metadata
- Advanced Topics

Metadata Overview

- Persist VOS types
  - Definitions (defs)
  - References (refs)
- Tool-to-Runtime
  - Class loader
  - Security
  - Implementation
- Tool-to-Tool
  - CLS (interoperability)
  - Specific tool data

Assemblies

Details next week

- *Assembly* is the unit of deployment
- Assemblies have *manifests* for resolving all cross-assembly references
- Manifests make assemblies self-describing and self-registering
- Manifests also specify policy for acceptance of upgraded external components

What's in the Metadata

- Define a type/method/field/property/event
- Reference a type/method/field
- Naming hierarchy:
  - Assembly lists its module files or other assembly files
  - Modules hold named classes, interfaces, properties, and events
  - Classes name methods and fields
  - Methods [illegible]
- Signatures (enforceable part of contracts)
- Map method definition to implementation
- Custom data

Signatures and Binding

- Metadata Engine converts refs to deal with single module (compile or link)
- Execution Engine binds refs to defs at class load time
- Compiler chooses exact signature or binding fails
- Extension mechanism under consideration for V1 to allow language-specific binding requirements
- V2 might allow language-specific fixups if binding fails

Access to Metadata

- Read, Write, Update, Enumerate, Merge
- Unmanaged APIs via COM-style calls
- Managed APIs via Microsoft.Runtime.Reflection
  - Enumerate
  - Create new types, methods, and fields
  - Instantiate
  - Persist

Advanced Topics

- Module-level functions and state
- Import and export across modules
- Declarative security
- Platform Invocation Services
  - Method information
  - Native calling convention
  - Native DLL and (mangled) name
  - Marshaling information (strings)

Agenda

- Type System
- Metadata
- Intermediate Language (IL)
  - http://msdn.microsoft.com/net/ECMA/Architecture.doc
  - http://msdn.microsoft.com/net/ECMA/CLIInstSet.doc
  - http://msdn.microsoft.com/net/ecma/cordlaunch/COM+ OPT-IL Specification.doc
- Just-In-Time Compilers (JITs)

Intermediate Language

- Introduction
- Overview
- IL By Category
- Floating Point Model
- Pointers and References
- Control Flow
- VOS Instructions
- OptIL
- Sample Program

Introduction

- Built for type safety
- Source language independent
- Architecture Independent (32-bit / 64-bit agnostic)
- RAD tools can target IL
- Optimizing compilers can target IL
- Stack based model
- Easy to generate verifiable code

Overview

- Stack Machine
- Automatic lifetime tracking for
- Exception tables
- Typed arguments and locals
- Stack types deducible
- Control flow restrictions
  - In/out of exception handlers
  - Backward branches with non-empty stack

IL By Category

218 instructions

- 42 arithmetic (including loading constants)
- 22 numeric conversion
- 12 annotations for optimization
- 31 stack access, args, local variables
- 44 control flow
- 14 Object Model
- 56 accessing and copying data

Floating Point Model

- Fixed size numbers in memory (4 byte, 8 byte, natural) and as parameters
- Loading onto stack expands to size convenient for target architecture
- Stack automatically widens in a target-specific manner
- Storing from stack truncates to memory size
- No exceptions, but test for NaN

Floating Point Model (Proposed Extensions)

- Prefix on conversion instructions to access additional IEEE modes
- Katmai/3DNow support via class library and JIT support
- Direct access to representation of floating point number

Control Flow

- Conditional and unconditional branches
- Switch
- Method call
  - Known method
  - Computed from object (virtual, interface)
  - General computation (function pointer)
- Tail call and jump
- Exception (throw, catch, filter, finally, leave)

Pointers and References

- Objects are opaque
- Objects must be reported to GC and may move
- Pointers must not point to objects and needn't be reported to GC
- By-ref parameters and locals must be reported to GC if they might point into an object
- By-ref values are *not* allowed as statics or on the GC heap

VOS Instructions

- Box and unbox
- Allocate/initialize object or array
- Typed references
- Cast and InstanceOf
- Critical regions
- Array operations
- Load and store fields
- Copy objects

OptIL

- Normal IL + Annotations
- Register allocate for pseudo-x86
- Interference graph for locals
  - Suggest promotion to registers
  - Priority ordering
- Simple trees only (finite state compiler)
  - Both depth and total size limit
- No nested calls
- SSA graph can be embedded

Sample Code (works)

Sample Driver

```
.global void main(class Microsoft.Runtime.String[])
.entrypoint
.maxstack 2
   ldsfld class Microsoft.Runtime.IO.W[illegible] Microsoft.Runtime.Text::Out
   ldarg.0          // Arg
   ldc.i4.0
   ldelem.ref       // Arg [0]
   call int32 Microsoft.Runtime.[illegible]::ToInteger(
                    class Microsoft.Runtime.String)
   call int32 [illegible]actorial(int32)
   callvirt void Microsoft.Runtime.IO.Writer::WriteLine(int32)
.end global
```

Agenda

- Type System
- Metadata
- Intermediate Language (IL)
- [illegible]

Just-In-Time Compilers

- Client-side Compilation is more accurate
- No interpreter
- Plan: Four choices
  - JIT (optimizing IL compiler)
  - EconoJIT (fast IL compiler)
  - OptJIT (fast OptIL compiler)
  - UTCJIT (superoptimizing IL compiler)
- Publish EE / JIT Interface
- OptIL
  - IL to OptIL is done by UTC (C++ back end)
  - Persist analysis and/or optimizations

EE / JIT Interface: Compile Time

- Method and field access
- Stub generation and runtime helper selection
- Interface pre-resolution
- JIT produces both code and data to management it (GC tables, EH tables, debugging information)
- Marshalling stub generation

EE / Code Interface: Run Time

- Runtime Helpers
  - Interface dispatch
  - Arithmetic helpers
  - Object Allocation
- Code manager
  - Walk back one frame
  - Find security data and "this" pointer
  - GC safe points
  - Report references to GC
  - Invoke exception filter or handler

Possible Future Topics

- Security
- Interop: COM and unmanaged APIs
- Class library
- Performance
- "You blew it" and "Have you thought about..."